МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и Структуры Данных»

Тема: Красно-черные деревья. Демонстрация.

Студент гр. 8304 _____ Мешков М.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2019

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Мешков М.А.

Группа 8304

Тема работы: Красно-черные деревья. Демонстрация.

Исходные данные:

Написать программу для демонстрации работы красно-черных деревьев поиска.

Содержание пояснительной записки:

- Содержание
- Введение
- Описание программы
- Тестирование
- Исходный код
- Использованные источники

Дата выдачи задания: 03.11.2019

Дата сдачи реферата:

Дата защиты реферата:

Студент _____ Мешков М.А.

Преподаватель _____ Фирсов М.А.

АННОТАЦИЯ

В данной работе была разработана программа на языке программирования C++ для демонстрации работы красно-черных бинарных деревьев поиска. Программа пошагово и подробно демонстрирует вставку и удаление элементов. Для лучшего понимания кода в исходном коде программы были оставлены комментарии. Была проведена работа по минимизации кода, улучшению быстродействия программы и сокращению потребления памяти.

SUMMARY

In this work, a program was developed in the C++ programming language to demonstrate the operation of red-black binary search trees. The program step by step and in detail demonstrates the insertion and removal of elements. For a better understanding of the code, comments were left in the source code of the program. Work was carried out to minimize the code, improve the performance of the program and reduce memory consumption.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является реализация красно-черного бинарного дерева поиска, демонстрация его работы. Красно-черные деревья являются одним из видов самобалансирующихся двоичных деревьев поиска, это значит, что они позволяют быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла. Эти свойства красно-черно дерева обеспечиваются введением дополнительного атрибута узла дерева — цвета. Это используется для организации сравнимых данных, таких как фрагменты текста или числа. Причем красно-чёрные деревья являются одними из наиболее активно используемых на практике самобалансирующихся деревьев поиска, они используются во многих языках программирования для реализации ассоциативного массива.

1. ОПИСАНИЕ КРАСНО-ЧЕРНЫХ ДЕРЕВЬЕВ

1.1. Принципы организации и работы

Красно-чёрное дерево представляют собой двоичное дерево поиска, в котором каждый узел имеет цвет. При этом оно обладает свойствами:

1. Узел может быть либо красным, либо чёрным и имеет двух потомков;
2. Корень — чёрный;
3. Все листья — чёрные и не содержат данных.
4. Оба потомка каждого красного узла — чёрные.
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Листовые узлы красно-чёрных деревьев не содержат данных и поэтому не требуют выделения памяти — достаточно записать в узле-предке в качестве указателя на потомка нулевой указатель. Однако, в некоторых реализациях могут использоваться явные листовые узлы, которые не содержат данных и просто служат для указания того, где дерево заканчивается. В данной работе используется представление листовых узлов через нулевые указатели.

Из этих свойств следует то, что путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего и дерево примерно сбалансировано. Это позволяет красно-чёрным деревьям быть более эффективными в худшем случае, чем обычным двоичным деревьям поиска.

Также красно-черные деревья более удобны для практического использования, чем идеально сбалансированные деревья, так как в последних тратится слишком много ресурсов на поддержание необходимой сбалансированности.

Поиск, вставка и удаление в красно-черных деревьях имеет сложность O(log n), как в среднем, так и в худшем случае.

1.2. Вставка элемента в дерево

Вставка значения в красно-черное дерево начинается с создания узла красного цвета и добавления его в соответствующее место в дереве вместо одного из листьев.

При вставки элемента в дерево (как и при удалении) могут нарушиться только свойства 2, 4, 5. Эти нарушения свойств необходимо исправить.

Для исправления нарушений свойств проверяются последовательно следующие случаи:

1. Текущий узел в корне дерева. В этом случае он просто перекрашивается в черный цвет и дерево становится корректным.
2. Предок текущего узла черный. В этом случае ни одно из свойств не нарушается, никаких дополнительных действий не требуется.
3. Родитель и дядя — красные. В этом случае родитель и дядя перекрашиваются в черный, а дедушка в красный. Однако теперь дедушка может нарушить свойства, поэтому начинается проверка заново всех случаев относительно дедушки.
4. Родитель является красным, а дядя черным. 2 шага:
    1. Осуществляется подготовка ко второму шагу: осуществляется поворот вокруг родителя так, чтобы текущий узел и узел-родитель были оба либо левыми детьми своих родителей, либо правыми. В случае осуществления поворота дальше идет рассмотрение относительно бывшего родителя.
    2. Осуществляется поворот вокруг дедушки, так чтобы родитель встал на его место. Цвета родителя и дедушки меняются. После этого все свойства выполняются.

1.3. Удаление элемента из дерева

Удаление элемента из красно-черного дерева выполняется сложнее, чем вставка в него.

Необходимо вначале свести случай удаления к удалению узла с одним ребенком или без детей, а затем удалить этот элемент, алгоритм следующий:

1. Если у N двое детей, то мы находится наибольший элемент D в левом поддереве; у него нет правого ребенка. Теперь можно скопировать данные из вершины D в вершину N, а саму вершину D удалить выше описанным способом. Далее вершиной N считается вершина D.
2. Если у вершины N нет детей, для удаления N достаточно поместить листовой узел вместо N.
3. Если у N один ребенок, то можно вырезать N, соединив его родителя напрямую с его ребенком.

Если вершина N была черная, то надо восстановить свойства красно-черного дерева (при удалении красной вершины свойства дерева не нарушаются). Пусть вершина K — ребенок удаленной вершины N. Если вершина K красная, то ее просто нужно перекрасить в черный и все свойства будут выполняться. Иначе возможны случаи (проверяются последовательно):

1. Вершина K — корень дерева. В этом случае все свойства выполняются, больше ничего делать не нужно.
2. Если у вершины K брат красного цвета, то делается поворот (брат становится родителем отца), при этом брат красится в черный, а отец в красный.
3. Если у вершины K родитель, его брат и его дети черного цвета, то брат окрашивается в красный цвет и далее работа происходит только с родителем начиная с первого случая.
4. Если у вершины K родитель красный, а брат и его дети черного цвета, то брат окрашивается в красный цвет, а родитель — в черный.

5. Если брат черный (он должен быть черным на этом шаге), то:
    1. Если K — левый ребенок, правый ребенок брата — черный, левый ребенок брата — красный, то цвет брата меняется на красный, а цвет левого ребенка брата на черный и делается правый поворот по брату.
    2. Если K — правый ребенок, левый ребенок брата — черный, правый ребенок брата — красный, то цвет брата меняется на красный, а цвет правого ребенка брата на черный и делается левый поворот по брату.
6. Брат перекрашивается в цвет родителя, а родитель - в черный. Далее:
    1. Если K — левый ребенок, то правый ребенок брата перекрашивается в черный и делается левый поворот по родителю.
    2. Если K — правый ребенок, то левый ребенок брата перекрашивается в черный и делается правый поворот по родителю.

После этого все свойства будут красно-черного дерева будут выполняться, а элемент будет успешно удален из дерева.

2. ОПИСАНИЕ ИСХОДНОГО КОДА

2.1. Файл main.cpp

В файле main.cpp (исходный код см. в приложении А) инициализируется создание всех используемых структур данных, она также отвечает за взаимодействие с пользователем (вывод сообщения-приглашения для ввода).

2.2. Структура RedBlackDemoTreeNode

Структура RedBlackDemoTreeNode представляет собой узел красно-черного дерева. Она содержит следующие поля: поле значение, указатели на левого ребенка, правого ребенка, указатель на родителя, переменную-перечисления для хранения цвета.

Она также содержит ряд методов для облегчения работы с перекрашиванием и определением цвета узла.

2.3. Класс RedBlackDemoTree

Класс RedBlackDemoTree реализует демонстрационное красно-черное бинарное дерево поиска. Оно содержит набор public методов для управления им:

- insert — позволяет вставить новое значение в дерево,
- remove — позволяет удалить значение из дерева,
- contains — позволяет узнать есть ли в дереве данное значение.

Также данный класс содержит набор private методов, реализующих внутреннюю логику класса.

Этот класс в качестве полей содержит указатель на корень дерева и на экземпляр класса RedBlackTreeHtmlDemonstrator.

2.4. Класс RedBlackTreeHtmlDemonstrator

Класс RedBlackTreeHtmlDemonstrator обеспечивает формирование HTML файла, в котором содержится сама демонстрация вставки и удаления дерева из узла. Этот класс использует файл redblacktreedemohtmltemplate.h для формирования результата.

2.5. Файл redblacktreedemohtmltemplate.h

Файл redblacktreedemohtmltemplate.h содержит две текстовых константы, содержащие шаблонные части HTML файла, в них содержатся определения стилей и скрипт для соединения узлов дерева линией.

3. ИНТЕРФЕЙС ПРОГРАММЫ

3.1. Интерфейс командной строки

Ввод в программу происходит с помощью интерфейса командной строки. Программа при запуске информирует пользователя о синтаксисе ввода (см. рис. 1).

```
This program waits to enter following possible actions:
'i' - insert value to the red-black tree (requires number),
'r' - remove value from the red-black tree (requires number).
The program will close if you enter incorrectly.
Example: i7 i 8 r7 exit
The result will be written to the file red-black-tree-demo.html
Enter actions:
i7 r7 end

Inserting value 7
Red-black tree properties validation: ok


Removing value 7
Red-black tree properties validation: ok
```

Рисунок 1 — Интерфейс командной строки

Этот интерфейс позволяет указать элементы для вставки и удаления. Также в нем присутствует информирование пользователя о результатах самопроверки программы (программа осуществляет контроль правильности всех свойств красно-черного дерева после каждой вставки и удаления).

3.2. Графический интерфейс

Графический интерфейс представляет собой результат работы программы, он представляет собой HTML файл, открытый в браузере для просмотра. Он содержит подробной описание и графическое представление процесса вставки и удаления (см. рис. 2).

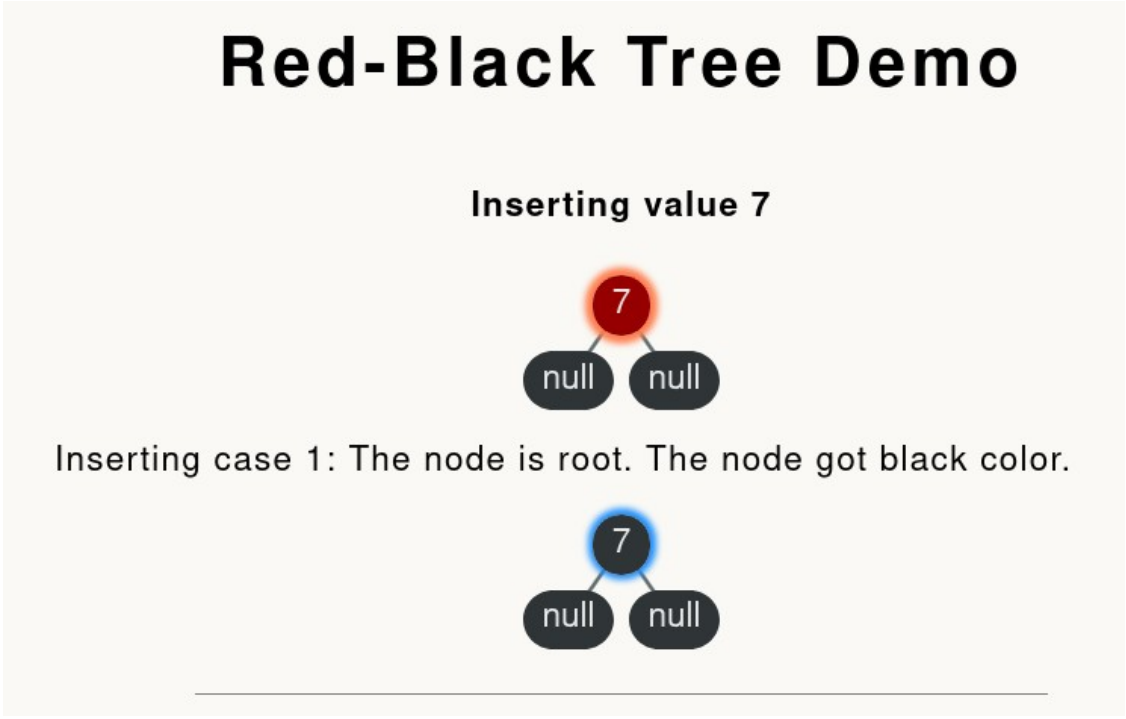

Рисунок 2 — Графический интерфейс программы

4. ТЕСТИРОВАНИЕ

Программа была протестирована на нескольких наборах данных (см. табл. 1).

Таблица 1 — Данные для тестирования программы

i0 i12 i13 i18 r13 i20 r12 i2 i7 r0 end
i3 i1 i4 i50 i7 r1 r7 r4 r50 r3 i1 i2 i4 r1 end
i3 i88 i0 i9 i3 i89 i21 i22 i23 i24 i25 i26 i27 i9 i1 i8 i64 i10 i1 r0 r9 r36 r26 r25 r24 r22 r10 r88 end
i10 i1000 i-1 i24 i 89 r78 r10 i102 i56 i34 i82 i94 i13 i35 i25 i29 i64 i31 i652 i46 r102 r-1 r89 r64 r31 r652 r46 i987 i31 i48 i156 i487 i25 i487 r 31 r156 r487 r25 quit
i59 i16 i17 i18 i19 i20 i21 i22 i23 i64 i65 i66 i67 i68 i69 i70 i1 i2 i3 i4 i5 i6 i7 i8 i9 i10 i11 i12 i13 i14 i15 i33 i24 i25 i26 i27 i28 i29 i30 i31 i32i60 i61 i62 i63 i34 i35 i36 i37 i38 i39 i40 i41 i42 i43 i44 i45 i46 i47 i48 i49 i50 r1 r2 r3 r4 r5 r6 r7 r8 r9 r41r48 r49 r50 r51 r52 r53 r54 r55 r56 r57 r58 r17 r18 r45 r46 r47 r13 r69r35r2 r65 r44 r61r14 r15 r16r20 r21 r22 r23 r24 r25 r26 r27 r28 r29 r30 r7 r70 r8 r9 r10 r11 r12r37 r38 r39 r40 r36 r67 r68r3r1 r19 r59 r60 r66 r31 r32 r33 r34 r4 r5 r6 r42 r43 r62 r63 r64 kjhkj

Тестирование производилось с помощью скрипта test.py (см. приложение Б), который был написан на языке программирования Python.

Вывод интерфейса командной строки во всех тестах свидетельствует об успешной проверке программы самой себя.

Также графический интерфейс пользователя показывает верные перестроения дерева.

ЗАКЛЮЧЕНИЕ

В ходе работы была написана программа для демонстрации работы красно-черного бинарного дерева поиска. Был получен опыт реализации бинарного дерева поиска на языке программирования C++. Были закреплены знания, полученные на протяжении семестра.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp

```
#include <iostream>

#include "redblackdemotree.h"

int main()
{
    RedBlackDemoTree tree;
    auto demonstrator = tree.getHtmlDemonstrator();

    std::cout
        << "This program waits to enter following possible actions:"
<< std::endl
        << "'i' - insert value to the red-black tree (requires
number)," << std::endl
        << "'r' - remove value from the red-black tree (requires
number)." << std::endl
        << "The program will close if you enter incorrectly." <<
std::endl
        << "Example: i7 i 8 r7 exit" << std::endl
        << "The result will be written to the file red-black-tree-
demo.html" << std::endl
        << "Enter actions: " << std::endl;

    char action;
    int number;
    while (true) {
        if (!(std::cin >> action && (action == 'i' || action == 'r')))
            break;
        if (!(std::cin >> number))
            break;
        std::cout << std::endl;

        if (action == 'i') {
            if (!tree.contains(number)) {
                std::string message = "Inserting value " +
std::to_string(number);
                std::cout << message << std::endl;
                demonstrator->printHeading(message);
                tree.insert(number);
            }
            else {
                std::string message = "Rejecting value " +
std::to_string(number) + " because it already exists.";
                std::cout << message;
                demonstrator->printHeading(message);
            }
        } else { // action = 'r'
```

```
            std::string message = "Removing value " +
std::to_string(number);
            std::cout << message << std::endl;
            demonstrator->printHeading(message);
            tree.remove(number);
        }

        std::cout << std::endl;
        demonstrator->printDivisor();
        demonstrator->writeToFile("red-black-tree-demo.html");
    }
    return 0;
}
```

Файл redblackdemotree.cpp

```
#include "redblackdemotree.h"

#include <iostream>
#include <cassert>
#include <string>

RedBlackDemoTree::RedBlackDemoTree()
    : m_htmlDemonstrator(new RedBlackTreeHtmlDemonstrator)
{}

RedBlackDemoTree::~RedBlackDemoTree() {
    erase(m_root);
    delete m_htmlDemonstrator;
}

RedBlackTreeHtmlDemonstrator *RedBlackDemoTree::getHtmlDemonstrator()
const {
    return m_htmlDemonstrator;
}

void RedBlackDemoTree::insert(const RedBlackDemoTree::ValueType
&value) {
    auto insertedNode = insertStupidly(value);
    insertCase1(insertedNode);
    validateProperties();
}

void RedBlackDemoTree::remove(const RedBlackDemoTree::ValueType
&value) {
    auto node = findNode(value);
    if (node == nullptr) {
        m_htmlDemonstrator->printTree(m_root);
        m_htmlDemonstrator->printLine("Nothing to remove. Do
nothing.");
        return;
    }
    m_htmlDemonstrator->highlightNode(node);
```

```
    m_htmlDemonstrator->printTree(m_root);
    node = goToOneChildCase(node);
    replaceNodeByChild(node);
    validateProperties();
}

void RedBlackDemoTree::erase(RedBlackDemoTree::Node *node) {
    if (node == nullptr)
        return;
    erase(node->left);
    erase(node->right);
    delete node;
}

RedBlackDemoTree::Node *RedBlackDemoTree::findNode(const
RedBlackDemoTree::ValueType &value) const {
    auto current = m_root;
    while (true) {
        if (current == nullptr)
            return nullptr;

        if (value > current->value)
            current = current->right;
        else if (value < current->value)
            current = current->left;
        else
            return current;
    }
}

RedBlackDemoTree::Node
*RedBlackDemoTree::getParent(RedBlackDemoTree::Node *node) {
    return node == nullptr ? nullptr : node->parent;
}

RedBlackDemoTree::Node
*RedBlackDemoTree::getGrandParent(RedBlackDemoTree::Node *node) {
    return getParent(getParent(node));
}

RedBlackDemoTree::Node
*RedBlackDemoTree::getSibling(RedBlackDemoTree::Node *node) {
    auto p = getParent(node);
    if (p == nullptr)
        return nullptr;
    if (node == p->left)
        return p->right;
    else
        return p->left;
}
```

```
RedBlackDemoTree::Node
*RedBlackDemoTree::getUncle(RedBlackDemoTree::Node *node) {
    return getSibling(getParent(node));
}

RedBlackDemoTree::Node
**RedBlackDemoTree::getNodeReferencePlace(RedBlackDemoTree::Node
*node) {
    assert(node != nullptr);
    if (node->parent != nullptr) {
        if (node->parent->left == node)
            return &node->parent->left;
        else
            return &node->parent->right;
    }
    else {
        return &m_root;
    }
}

void RedBlackDemoTree::rotateLeft(RedBlackDemoTree::Node *node) {
    assert(node != nullptr);
    auto pivot = node->right;
    assert(pivot != nullptr);

    pivot->parent = node->parent;
    *getNodeReferencePlace(node) = pivot;

    node->right = pivot->left;
    if (pivot->left != nullptr)
        pivot->left->parent = node;

    node->parent = pivot;
    pivot->left = node;

    m_htmlDemonstrator->highlightNodes({node, pivot});
}

void RedBlackDemoTree::rotateRight(RedBlackDemoTree::Node *node) {
    assert(node != nullptr);
    auto pivot = node->left;
    assert(pivot != nullptr);

    pivot->parent = node->parent;
    *getNodeReferencePlace(node) = pivot;

    node->left = pivot->right;
    if (pivot->right != nullptr)
        pivot->right->parent = node;

    node->parent = pivot;
    pivot->right = node;
```

```
    m_htmlDemonstrator->highlightNodes({node, pivot});
}

RedBlackDemoTree::Node *RedBlackDemoTree::insertStupidly(const
RedBlackDemoTree::ValueType &value) {
    auto node = new Node;
    node->value = value;

    Node *prevCurrent = nullptr;
    Node **current = &m_root;
    while (true) {
        if (*current == nullptr) {
            *current = node;
            node->parent = prevCurrent;
            break;
        }
        prevCurrent = *current;
        if (value > (*current)->value)
            current = &(*current)->right;
        else
            current = &(*current)->left;
    }

    m_htmlDemonstrator->highlightNode(node);
    m_htmlDemonstrator->printTree(m_root);

    return node;
}

void RedBlackDemoTree::insertCase1(RedBlackDemoTree::Node *node) {
    if (node->parent == nullptr) {
        // Node has no parent
        node->recolorToBlack();
        m_htmlDemonstrator->printLine("Inserting case 1: The node is
root. The node got black color.");
        m_htmlDemonstrator->highlightNode(node);
        m_htmlDemonstrator->printTree(m_root);
    }
    else {
        // Node has parent
        insertCase2(node);
    }
}

void RedBlackDemoTree::insertCase2(RedBlackDemoTree::Node *node) {
    if (node->parent->isBlack()) {
        // Node has black parent
        // Do nothing.
        m_htmlDemonstrator->printLine("Inserting case 2: The node has
black parent. Do nothing.");
        return;
```

```
    }
    else {
        // Node has red parent
        insertCase3(node);
    }
}

void RedBlackDemoTree::insertCase3(RedBlackDemoTree::Node *node) {
    auto u = getUncle(node);

    if (u != nullptr && u->isRed()) {
        // Node has red parent and red uncle
        node->parent->recolorToBlack();
        u->recolorToBlack();
        auto g = getGrandParent(node);
        g->recolorToRed();

        m_htmlDemonstrator->printLine("Inserting case 3: The node has
red parent and red uncle. "
                                      "The uncle and the parent is
recolored to black, the grandparent is recolored to red. "
                                      "Start working with the
grandparent.");
        m_htmlDemonstrator->highlightNodes({node->parent, u, g});
        m_htmlDemonstrator->printTree(m_root);

        insertCase1(g);
    }
    else {
        // Node has red parent and black or no uncle
        insertCase4Step1(node);
    }
}

void RedBlackDemoTree::insertCase4Step1(RedBlackDemoTree::Node *node)
{
    auto g = getGrandParent(node);

    std::string message = "Inserting case 4 step 1: The node has red
parent and black or no uncle. "
                          "Preparation to step 2: ";

    if (node == node->parent->right && node->parent == g->left) {
        rotateLeft(node->parent);
        node = node->left;
        message += "Rotation to left around the parent.";
        m_htmlDemonstrator->printLine(message);
        m_htmlDemonstrator->printTree(m_root);
    }
    else if (node == node->parent->left && node->parent == g->right) {
        rotateRight(node->parent);
        node = node->right;
```

```
        message += "Rotation to right around the parent.";
        m_htmlDemonstrator->printLine(message);
        m_htmlDemonstrator->printTree(m_root);
    }
    else {
        message += "Do nothing.";
        m_htmlDemonstrator->printLine(message);
    }

    insertCase4Step2(node);
}

void RedBlackDemoTree::insertCase4Step2(RedBlackDemoTree::Node *node)
{
    auto g = getGrandParent(node);

    node->parent->recolorToBlack();
    g->recolorToRed();

    std::string message = "Inserting case 4 step 2: The parent
recolored to black, "
                          "the grandparent is recolored to red, ";

    if (node == node->parent->left) {
        rotateRight(g);
        message += "rotation to right around the grandparent.";
    }
    else {
        rotateLeft(g);
        message += "rotation to left around the grandparent.";
    }

    m_htmlDemonstrator->printLine(message);
    m_htmlDemonstrator->printTree(m_root);
}

RedBlackDemoTree::Node
*RedBlackDemoTree::findMaxInSubtree(RedBlackDemoTree::Node *node) {
    if (node == nullptr)
        return nullptr;
    while (true) {
        if (node->right == nullptr)
            return node;
        node = node->right;
    }
}

RedBlackDemoTree::Node
*RedBlackDemoTree::findMinInSubtree(RedBlackDemoTree::Node *node) {
    if (node == nullptr)
        return nullptr;
    while (true) {
```

```
        if (node->left == nullptr)
            return node;
        node = node->left;
    }
}

RedBlackDemoTree::Node
*RedBlackDemoTree::goToOneChildCase(RedBlackDemoTree::Node *node) {
    std::string message = "Going to one-child case: ";

    if (node->left == nullptr || node->right == nullptr) {
        message += "The node already doesn't have more than one child.
Do nothing.";
        m_htmlDemonstrator->printLine(message);
        return node;
    }
    // Here node->left != nullptr and node->right != nullptr

    message += "The node's value was replaced by its ";

    Node *descendant = findMaxInSubtree(node->left);
    message += "left max";

    node->value = descendant->value;

    message += " descendant's value. Start working with the
descendant.";
    m_htmlDemonstrator->printLine(message);
    m_htmlDemonstrator->highlightNodes({node, descendant});
    m_htmlDemonstrator->printTree(m_root);

    return descendant;
}

void RedBlackDemoTree::replaceNode(RedBlackDemoTree::Node *node,
RedBlackDemoTree::Node *replacement) {
    assert(node != nullptr);
    if (replacement != nullptr) {
        replacement->parent = node->parent;
    }
    *getNodeReferencePlace(node) = replacement;
    // Caller should delete "node"
}

void RedBlackDemoTree::replaceNodeByChild(RedBlackDemoTree::Node
*node) {
    std::string message = "Replacing the node by its child: ";

    auto child = (node->right == nullptr ? node->left : node->right);
    if (node->isBlack()) {
        message += "The node is black, ";
        if (isRed(child)) {
```

```
            child->recolorToBlack();
            message += "child is red. Recoloring the child to black
and replacing the node by the child.";
        }
        else { // child is black
            // Using "node" instead "child" (cause they both have
black color but child may be null.
            message += "child is black. Replacing was deferred.";
            m_htmlDemonstrator->printLine(message);
            removeCase1(node);
            message = "Replacing the node by its child (continuation):
Just replacing the node by the child.";
        }
    }
    else { // node is red
        message += "Node is red (=> the child is null). Just replacing
the node by the child.";
    }
    replaceNode(node, child);
    delete node;
    m_htmlDemonstrator->printLine(message);
    m_htmlDemonstrator->highlightNode(child);
    m_htmlDemonstrator->printTree(m_root);
}

void RedBlackDemoTree::removeCase1(RedBlackDemoTree::Node *node) {
    if (node->parent == nullptr) {
        // Node is root
        m_htmlDemonstrator->printLine("Delete case 1: The node is
root. Do nothing.");
    }
    else {
        // Node is no root
        removeCase2(node);
    }
}

void RedBlackDemoTree::removeCase2(RedBlackDemoTree::Node *node) {
    Node *s = getSibling(node);
    if (isRed(s)) {
        std::string message = "Delete case 2: The sibling is red. ";

        node->parent->recolorToRed();
        s->recolorToBlack();
        message += "Recoloring the parent to red and the sibling to
black, ";

        if (node == node->parent->left) {
            rotateLeft(node->parent);
            message += "rotation to left around the parent.";
        }
        else {
```

```
            rotateRight(node->parent);
            message += "rotation to right around the parent.";
        }

        m_htmlDemonstrator->printLine(message);
        m_htmlDemonstrator->highlightNodes({node->parent, s});
        m_htmlDemonstrator->printTree(m_root);
    }
    removeCase3(node);
}

void RedBlackDemoTree::removeCase3(RedBlackDemoTree::Node *node) {
    Node *s = getSibling(node);
    if (isBlack(node->parent) && isBlack(s) && isBlack(s->left) &&
isBlack(s->right)) {
        std::string message;
        message += "Remove case 3: The node's parent, the sibling, the
sibling' left child, "
                   "the sibling's right child is black. ";

        s->recolorToRed();
        message += "Recoloring the sibling to red. Start working with
the node's parent.";

        m_htmlDemonstrator->printLine(message);
        m_htmlDemonstrator->highlightNode(s);
        m_htmlDemonstrator->printTree(m_root);
        removeCase1(node->parent);
    }
    else {
        removeCase4(node);
    }
}

void RedBlackDemoTree::removeCase4(RedBlackDemoTree::Node *node) {
    Node *s = getSibling(node);
    if (isRed(node->parent) && isBlack(s) && isBlack(s->left) &&
isBlack(s->right)) {
        std::string message;
        message += "Remove case 4: The node's parent is red but "
                   "the sibling, the sibling' left child, the
sibling's right child is black. ";

        s->recolorToRed();
        node->parent->recolorToBlack();
        message += "Recoloring the sibling to red and the node's
parent to black.";

        m_htmlDemonstrator->printLine(message);
        m_htmlDemonstrator->highlightNodes({s, node->parent});
        m_htmlDemonstrator->printTree(m_root);
    }
```

```
    else {
        removeCase5(node);
    }
}

void RedBlackDemoTree::removeCase5(RedBlackDemoTree::Node *node) {
    Node *s = getSibling(node);
    if (isBlack(s)) {
        std::string message;
        message += "Remove case 5: The sibling is black, ";

        if ((node == node->parent->left) && isBlack(s->right) &&
isRed(s->left)) {
            message += "the node is the left child of its parent, the
sibling's right child is black "
                       "and the sibling's left child is red. ";

            s->recolorToRed();
            s->left->recolorToBlack();
            rotateRight(s);
            message += "Recoloring the sibling to red, the sibling's
left child to black, "
                       "rotation right around the sibling.";
        }
        else if ((node == node->parent->right) && isBlack(s->left) &&
isRed(s->right)) {
            message += "the node is the right child of its parent, the
sibling's left child is black "
                       "and the sibling's right child is red. ";

            s->recolorToRed();
            s->right->recolorToBlack();
            rotateLeft(s);
            message += "Recoloring the sibling to red, the sibling's
right child to black, "
                       "rotation left around the sibling.";
        }
    }
    removeCase6(node);
}

void RedBlackDemoTree::removeCase6(RedBlackDemoTree::Node *node) {
    Node *s = getSibling(node);

    std::string message = "Remove case 6: ";

    s->color = node->parent->color;
    node->parent->recolorToBlack();
    message += "Recoloring sibling's color to the color of the node's
parent,"
               "recoloring the node's parent to black. ";
```

```
    if (node == node->parent->left) {
        s->right->recolorToBlack();
        rotateLeft(node->parent);
        message += "Recoloring the sibling's right child to black,
rotation to left around the node's parent.";
    }
    else {
        s->left->recolorToBlack();
        rotateRight(node->parent);
        message += "Recoloring the sibling's left child to black,
rotation to right around the node's parent.";
    }
}

void RedBlackDemoTree::validateProperties() {
#ifndef NDEBUG
    if (m_root != nullptr) {
        assert(m_root->isBlack());
        validateColors(m_root);
        validatePathLengths(m_root);
    }

    std::cout << "Red-black tree properties validation: ok" <<
std::endl;
#endif
}

size_t RedBlackDemoTree::validatePathLengths(RedBlackDemoTree::Node
*node) {
    if (node == nullptr)
        return 1;
    auto len = validatePathLengths(node->left);
    assert(len == validatePathLengths(node->right));
    if (node->isBlack())
        return len + 1;
    else
        return len;
}

void RedBlackDemoTree::validateColors(RedBlackDemoTree::Node *node) {
    if (node == nullptr)
        return;
    if (node->isRed()) {
        assert(node->left == nullptr || node->left->isBlack());
        assert(node->right == nullptr || node->right->isBlack());
    }
    validateColors(node->left);
    validateColors(node->right);
}
```

Файл redblackdemotree.h

```
#pragma once
```

```
#include "redblacktreehtmldemonstrator.h"

struct RedBlackDemoTreeNode {
    int value = 0;
    RedBlackDemoTreeNode *parent = nullptr, *left = nullptr, *right =
nullptr;
    enum class Color : bool {
        BLACK,
        RED
    } color = Color::RED;

    inline bool isBlack() const { return color == Color::BLACK; };
    inline bool isRed() const { return color == Color::RED; };
    inline void recolorToBlack() { color = Color::BLACK; };
    inline void recolorToRed() { color = Color::RED; };
};

class RedBlackDemoTree {
public:
    RedBlackDemoTree();
    RedBlackDemoTree(const RedBlackDemoTree &) = delete;
    ~RedBlackDemoTree();
    RedBlackDemoTree &operator=(const RedBlackDemoTree &) = delete;

    RedBlackTreeHtmlDemonstrator *getHtmlDemonstrator() const;

    using ValueType = int;

    void insert(const ValueType &value);
    void remove(const ValueType &value);
    inline bool contains(const ValueType &value) const {
        return findNode(value) != nullptr;
    }

private:
    using Node = RedBlackDemoTreeNode;
    Node *m_root = nullptr;
    RedBlackTreeHtmlDemonstrator *m_htmlDemonstrator;

    static void erase(Node *node);
    Node *findNode(const ValueType &value) const;

    static Node *getParent(Node *node);
    static Node *getGrandParent(Node *node);
    static Node *getSibling(Node *node);
    static Node *getUncle(Node *node);
    Node **getNodeReferencePlace(Node *node);
    void rotateLeft(Node *node);
    void rotateRight(Node *node);

    [[nodiscard]] Node *insertStupidly(const ValueType &value);
    void insertCase1(Node *node);
```

```
    void insertCase2(Node *node);
    void insertCase3(Node *node);
    void insertCase4Step1(Node *node);
    void insertCase4Step2(Node *node);

    static Node *findMaxInSubtree(Node *node);
    static Node *findMinInSubtree(Node *node);
    Node *goToOneChildCase(Node *node);
    void replaceNode(Node *node, Node *replacement);
    static inline bool isBlack(Node *node) {
        return node == nullptr || node->isBlack();
    }
    static inline bool isRed(Node *node) {
        return !isBlack(node);
    }
    void replaceNodeByChild(Node *node);
    void removeCase1(Node *node);
    void removeCase2(Node *node);
    void removeCase3(Node *node);
    void removeCase4(Node *node);
    void removeCase5(Node *node);
    void removeCase6(Node *node);

    void validateProperties();
    static void validateColors(Node *node);
    static size_t validatePathLengths(Node *node);
};
```

Файл redblacktreedemohtmltemplate.h

```
#pragma once

constexpr const char *redBlackTreeDemoHtmlTemplateTopPart = R"(
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1">
    <title>Red-Black Tree Demo</title>
    <style>
      html {
        background-color: #faf9f5;
        font-family: sans-serif;
      }
      body {
        /* max-width: 600px; */
        margin: 0 auto;
        padding: 0 20px 20px;
      }
      h1 {
        text-align: center;
        letter-spacing: 3px;
```

```
  margin: 0;
  padding: 30px 0;
}
.red-node, .black-node {
  padding: 5px 10px;
  margin: 4px;
  border-radius: 100px; // TOFIX
}
.red-node {
  background-color: rgb(150,0,0);
  color: rgba(255, 255, 255, 0.9);
}
.black-node {
  background-color: #2f3437;
  color: rgba(255, 255, 255, 0.9);
}
.black-node.highlighted {
  box-shadow: 0 0 3px 3px dodgerblue;
}
.red-node.highlighted {
  box-shadow: 0 0 3px 4px coral;
}
.tree, .root, .children, .red-node, .black-node {
  display: inline-block;
}
.root {
  display: flex;
  justify-content: center;
}
.root > * {
  flex-grow: 0;
}
.children {
  display: flex;
  justify-content: center;
  align-items: flex-start;
}
.children > * {
  flex-grow: 0;
}
.tree-holder {
  position: relative;
  padding: 10px;
  text-align:center;
}
.tree-holder > canvas {
  position: absolute;
  z-index: -1;
}
.text, .heading {
  line-height: 1.5;
  letter-spacing: 1px;
```

```
        max-width: 600px;
        margin: 0 auto;
      }
      hr {
        width: 75%;
        border-top-width: 0px;
        max-width: calc(600px * 0.75);
      }
      .heading {
        text-align: center;
        font-weight: bold;
        margin: 8px auto 8px;
      }
    </style>
  </head>
  <body>
    <h1>Red-Black Tree Demo</h1>
)";

constexpr const char *redBlackTreeDemoHtmlTemplateBottomPart = R"(
  <script>
    let treeHolders = document.getElementsByClassName("tree-holder");
    for (let treeHolder of treeHolders) {
      let canvas = document.createElement("canvas")
      treeHolder.insertBefore(canvas, treeHolder.firstChild)
      let context = canvas.getContext('2d')

      let mainTree = treeHolder.querySelector(":scope > .tree")
      if (mainTree == null)
        continue
      let dpr = window.devicePixelRatio || 1 // fallback to 1
      canvas.style.width = mainTree.offsetWidth + 'px'
      canvas.style.height = mainTree.offsetHeight + 'px'
      canvas.width = mainTree.offsetWidth * dpr
      canvas.height = mainTree.offsetHeight * dpr
      context.scale(dpr, dpr)

      context.lineWidth = 2
      context.strokeStyle = "#5f6d70"

      let trees = treeHolder.querySelectorAll(".tree")
      for (let tree of trees) {
        let root = tree.querySelector(":scope
> .root").firstElementChild
        let children = tree.querySelector(":scope
> .children").children

        let canvasX = canvas.getBoundingClientRect().left
        let canvasY = canvas.getBoundingClientRect().top
        let rootX = root.getBoundingClientRect().left +
root.offsetWidth / 2
```

```
        let rootY = root.getBoundingClientRect().top +
root.offsetHeight / 2

        for (var child of children) {
          if (child.classList.contains("tree")) {
            child = child.querySelector(":scope
> .root").firstElementChild
          }
          let childX = child.getBoundingClientRect().left +
child.offsetWidth / 2
          let childY = child.getBoundingClientRect().top +
child.offsetHeight / 2

          context.beginPath()
          context.moveTo(rootX - canvasX, rootY - canvasY)
          context.lineTo(childX - canvasX, childY - canvasY)
          context.closePath()
          context.stroke()
        }
      }
    }
  </script>

  </body>
</html>
)";
```

Файл redblacktreehtmldemonstrator.cpp

```
#include "redblacktreehtmldemonstrator.h"

#include "redblackdemotree.h"
#include "redblacktreedemohtmltemplate.h"

#include <fstream>
#include <iostream>

void RedBlackTreeHtmlDemonstrator::printLine(const std::string &line)
{
    m_htmlContent << "\n";
    m_htmlContent << "<div class=text>" << line << "</div>\n";
}

void RedBlackTreeHtmlDemonstrator::printDivisor() {
    m_htmlContent << "<hr>\n";
}

void RedBlackTreeHtmlDemonstrator::printHeading(const std::string
&heading) {
    m_htmlContent << "\n";
    m_htmlContent << "<div class=heading>" << heading << "</div>\n";
}
```

```
void
RedBlackTreeHtmlDemonstrator::printTreeWithoutHolder(RedBlackDemoTreeN
ode *root) {
    if (root == nullptr) {
        m_htmlContent << "<div class=black-node>null</div>\n";
        return;
    }

    m_htmlContent << "<div class=tree>\n";

    m_htmlContent << "<div class=root>\n";
    m_htmlContent << std::string("<div class=\"")
                  << (root->isRed() ? "red-node" : "black-node")
                  << (m_toHighlight.count(root) != 0 ? "
highlighted" : "")
                  << "\">" << root->value << "</div>\n";
    m_htmlContent << "</div>\n"; // Closing <div class=root>

    m_htmlContent << "<div class=children>\n";
    printTreeWithoutHolder(root->left);
    printTreeWithoutHolder(root->right);
    m_htmlContent << "</div>\n"; // Closing <div class=children>

    m_htmlContent << "</div>\n"; // Closing <div class=tree>
}

void RedBlackTreeHtmlDemonstrator::highlightNode(RedBlackDemoTreeNode
*node) {
    m_toHighlight.insert(node);
}

void
RedBlackTreeHtmlDemonstrator::highlightNodes(std::initializer_list<Red
BlackDemoTreeNode *> nodes) {
    m_toHighlight.insert(nodes);
}


void RedBlackTreeHtmlDemonstrator::printTree(RedBlackDemoTreeNode
*root) {
    m_htmlContent << "\n";
    m_htmlContent << "<div class=tree-holder>\n";
    printTreeWithoutHolder(root);
    m_htmlContent << "</div>\n";
    m_toHighlight.clear();
}

void RedBlackTreeHtmlDemonstrator::writeToFile(const std::string
&path) const {
    std::ofstream file;
    file.open(path);
    if (file.is_open()) {
```

```
        file << redBlackTreeDemoHtmlTemplateTopPart;
        file << m_htmlContent.str();
        file << redBlackTreeDemoHtmlTemplateBottomPart;
        file.close();
    }
    else {
        std::cerr << "Error: file opening is failed." << std::endl;
    }
}
```

Файл redblacktreehtmldemonstrator.h

```
#pragma once

#include <string>
#include <sstream>
#include <set>

class RedBlackDemoTreeNode;

class RedBlackTreeHtmlDemonstrator {
public:
    void printLine(const std::string &line);
    void printDivisor();
    void printHeading(const std::string &heading);

    void highlightNode(RedBlackDemoTreeNode *node);
    void highlightNodes(std::initializer_list<RedBlackDemoTreeNode *>
nodes);
    void printTree(RedBlackDemoTreeNode *root);

    void writeToFile(const std::string &path) const;

private:
    std::ostringstream m_htmlContent;
    std::set<RedBlackDemoTreeNode *> m_toHighlight;

    void printTreeWithoutHolder(RedBlackDemoTreeNode *root);
};
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД ТЕСТОВОГО СКРИПТА

```
import subprocess
from subprocess import Popen
import os

program = './course_work'

if not os.path.isfile(program):
    print(program, "is not found.")
    quit()

def test_with_file(path):
    with open(path, 'r') as file:
        iteration = 0
        while True:
            iteration += 1

            def readline():
                line = file.readline()
                if not line:
                    return None
                return line.rstrip('\n') + '\n'

            input = readline()
            if not input:
                break

            p = Popen(program, stdin=subprocess.PIPE,
stdout=subprocess.PIPE, stderr=subprocess.STDOUT, text=True)

            out, _ = p.communicate(input)

            enter_pos = out.find('Enter actions:') + len('Enter
actions:\n')
            out = out[:enter_pos] + input + out[enter_pos:]

            print(f'Test {iteration}:')
            print(out)

            os.rename('red-black-tree-demo.html', f'red-black-tree-
demo-{iteration}.html')

for address, _, files in os.walk('Tests'):
    for file in files:
        path = os.path.join(address, file)
        print(path, ':', sep='')
        print()
        test_with_file(path)
```